UNCLASSIFIED

DTIC FILE COPY

AD-A207 029

ata Entered)

ION PAGE

	READ INSTRUCTIONS BEFORE COMPLETING FORM
12. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER

4. Ada Compiler Validation Summary Report: Tandem Computers, Tandem Ada, Version T9270C10, Tandem NonStop VLX (Host) and (Target), 880520W1.09060

5. TYPE OF REPORT & PERIOD COVERED
25 May 1988 to 25 May 1988

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Wright-Patterson AFB
Dayton, OH

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS

Wright-Patterson
Dayton, OH

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)

Wright-Patterson
Dayton, OH

15. SECURITY CLASS (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DTIC
ELECTE
13 APR 1989
E

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 If different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Tandem Computers, Tandem Ada, Version T9270C10, Wright-Patterson AFB, Tandem NonStop VLX under GUARDIAN 90, Version C10 (Host) to Tandem NonStop VLX under GUARDIAN 90, Version C10 (Target), ACVC 1.9.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

89 4 11 001

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 880520W1.09060
Tandem Computers
Tandem Ada, Version T9270C10
Tandem NonStop VLX

Completion of On-Site Testing:
25 May 1988

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

·Compiler Name: Tandem Ada, Version T9270C10

Certificate Number: 880520W1.09060

Host: Target:
    Tandem NonStop VLX under   Tandem NonStop VLX under
    GUARDIAN 90,   GUARDIAN 90,
    Version C10   Version C10

Testing Completed 25 May 1988 Using ACVC 1.9

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH   45433-6503

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA   22311

Ada Joint Program Office
William S. Ritchie
Acting Director
Department of Defense
Washington, DC 20301

Ada Compiler Validation Summary Report:

Compiler Name: Tandem Ada, Version T9270C10

Certificate Number: 880520W1.09060

Host:
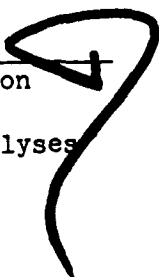 Tandem NonStop VLX under
  GUARDIAN 90,
  Version C10

Target:
  Tandem NonStop VLX under
  GUARDIAN 90,
  Version C10

Testing Completed 25 May 1988 Using ACVC 1.9

This report has been reviewed and is approved.

Steven P. Wilson
Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH   45433-6503

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA   22311

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC   20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at bind time, and during execution.

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard

- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 25 May 1988 at Cupertino, CA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

> Ada Information Clearinghouse
> Ada Joint Program Office
> OUSDRE
> The Pentagon, Rm 3D-139 (Fern Street)
> Washington DC 20301-3081

or from:

> Ada Validation Facility
> ASD/SCEL
> Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

> Ada Validation Organization
> Institute for Defense Analyses
> 1801 North Beauregard Street
> Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC   The Ada Compiler Validation Capability. The set of Ada programs that test the conformity of an Ada compiler to the Ada programming language.

Ada   An Ada Commentary contains all information relevant to the
Commentary point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.

Ada Standard ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant The agency requesting validation.

AVF   The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.

AVO   The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical

support for Ada validations to ensure consistent practices.

Compiler   A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

Failed test   An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host   The computer on which the compiler resides.

Inapplicable test   An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test   An ACVC test for which a compiler generates the expected result.

Target   The computer for which a compiler generates code.

Test   A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn test   An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce bind (link) errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at bind (link) time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and

place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Tandem Ada, Version T9270C10

ACVC Version: 1.9

Certificate Number: 880520W1.09060

Host Computer:

Machine:	Tandem NonStop VLX
Operating System:	GUARDIAN 90 Version C10
Memory Size:	8 Mbytes

Target Computer:

Machine:	Tandem NonStop VLX
Operating System:	GUARDIAN 90 Version C10
Memory Size:	8 Mbytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation supports the additional predefined types SHORT_INTEGER, LONG_INTEGER, LONG_FLOAT, and LONG_LONG_INTEGER in the package STANDARD. (See tests B86001C and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

  This implementation exhibits behavior suggesting that no default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

  Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

CONSTRAINT_ERROR is raised when a literal operand of type INTEGER in a comparison test is outside the range of the base type. No exception is raised when a literal operand of type INTEGER in a membership test is outside the range of the base type. For the largest integer type, NUMERIC_ERROR is raised when a literal operand in either a membership or comparison test is outside the range of the type. (See test C45232A.)

NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Underflow could not be tested for graduality. (See tests C45524A..Z and the section in this document on inapplicable tests.)

. Rounding.

This implementation exhibits behavior suggesting that the method used for rounding to integer and longest integer is round away from zero. (See tests C46012A..Z.)

This implementation exhibits behavior suggesting that the method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises CONSTRAINT_ERROR. (See test C36003A.)

No exception is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises CONSTRAINT_ERROR when the array objects are sliced. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

This implementation exhibits behavior suggesting that, in assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the exhibited behavior is that the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

This implementation exhibits behavior suggesting that, in assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Aggregates.

This implementation exhibits behavior suggesting that, in the evaluation of a multi-dimensional aggregate, all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

CONSTRAINT_ERROR is raised before all choices are evaluated when a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

. Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it. For this implementation:

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are not supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are not supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are not supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are not supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Record representation clauses are supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

. Pragmas.

The pragma INLINE is supported for procedures and functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

. Input/output.

The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO.   (See tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to last element written. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, cannot be created in OUT_FILE mode, and cannot be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text input/output for reading only. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each external file for sequential input/output for reading only. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct input/output for reading only. (See tests CE2107F..I (5 tests) and CE2110B)

An internal sequential access file and an internal direct access file cannot be associated with a single external file for writing. (See test CE2107E.)

Temporary sequential and direct files are given names.   Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

. Generics.

Separate compilation of generic bodies without specifications is not supported by this implementation; they are rejected at compile time. Compilation of a generic specification without its corresponding body is supported. (See tests CA1012A, CA2009C, CA2009F, CA3011A, ,LA5008M..N (2 tests), BC1011C, BC3204C, and BC3205D.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was
tested, 27 tests had been withdrawn because of test errors. The AVF
determined that 265 tests were inapplicable to this implementation. All
inapplicable tests were processed during validation testing except for 187
executable tests that use floating-point precision exceeding that supported
by the implementation. Modifications to the code, processing, or grading
for 53 tests were required to successfully demonstrate the test objective.
(See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable
conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	A	B	C	D	E	L	TOTAL
Passed	104	1049	1602	17	14	44	2830
Inapplicable	6	2	251	0	4	2	265
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	190	490	544	247	166	98	140	325	131	36	232	3	228	2830
Inapplicable	14	82	130	1	0	0	3	2	6	0	2	0	25	265
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C
C35904A	C35904B	C35A03E	C35A03R	C37213H
C37213J	C37215C	C37215E	C37215G	C37215H
C38102C	C41402A	C45332A	C45614C	A74106C
C85018B	C87B04B	CC1311B	BC3105A	AD1A01A
CE2401H	CE3208A			

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 265 tests were inapplicable for the reasons indicated:

- A28004A uses the pragma PRIORITY, which is not supported, and attempts to calculate PRIORTY'FIRST, where the predefined PRIORITY subtype is a null range.

- C35502I..J (2 tests), C35502M..N (2 tests), C35507I..J (2 tests), C35507M..N (2 tests), C35508I..J (2 tests), C35508M..N (2 tests), A39005F, and C55B16A use enumeration representation clauses which are not supported by this compiler.

- C35702A uses SHORT_FLOAT which is not supported by this implementation.

- A39005C and C87B62B use length clauses with STORAGE_SIZE specifications for access types which are not supported by this implementation.

- A39005G uses a record representation clause which is not supported by this implementation.

- C45504B and C45632B expect an intermediate SHORT_INTEGER value to raise an exception, which is not done by this implementation.

- C45524A..Z (26 tests) require that F'MACHINE_RADIX ** (F'MACHINE_EMIN - 1) be distinct from the representation from zero; however, the avo ruled that in light of the commentary AI-00543 the requirement need no be met.

- C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

- C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

- C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

- C87B62D and C92005B depend on the default STORAGE_SIZE of a task type, which for this implementation is 2 ** 18, out of the range of type INTEGER.

- C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.

- CA2009F and CA1012A compile generic subprogram declarations and bodies in separate compilations. This compiler does not support separate compilation of generic subprogram bodies.

- CA2009C, BC3204C, and BC3205D compile generic package specifications and bodies in separate compilations. This compiler does not support separate compilation of generic package bodies.

- CA3011A and LA5008M..N (2 tests) compile generic unit bodies and subunits in separate compilations. This compiler does not support separate compilation of generic unit bodies and their subunits.

- AE2101C, EE2201D, and EE2201E use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

- AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

- CE2105A..B (2 tests), CE2111H, and CE3109A are inapplicable because this implementation does not support CREATE with the MODE parameter IN_FILE.

- CE2107B..E, G..I (7 tests), CE2110B, CE2111D, CE3111B..E (4 tests), CE3114B, and CE3115A are inapplicable because multiple internal files cannot be associated with the same external file unless all files are opened for reading only. The proper exception is raised when multiple access is attempted.

- The following 173 tests require a floating-point accuracy that exceeds the maximum of 16 digits supported by this implementation:

C24113M..Y (13 tests)	C35707M..Y (13 tests)
C35706M..Y (13 tests)	C35802M..Z (14 tests)
C35708M..Y (13 tests)	C45321M..Y (13 tests)
C45241M..Y (13 tests)	C45521M..Z (14 tests)
C45421M..Y (13 tests)	C45621M..Z (14 tests)
C45641M..Y (13 tests)	C46012M..Z (14 tests)
C35705M..Y (13 tests)	

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 52 Class B tests and 1 Class C test.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B22004A	B22004B	B22004C	B23004A
B23004B	B24001A	B24001B	B24001C	B24005A
B24005B	B24007A	B24009A	B24204A	B24204B
B24204C	B26002A	B28003C	B29001A	B2A003A
B2A003B	B2A003C	B2A007A	B2A010A	B33301A

B35101A	B36002A	B36201A	B37201A	B37307B
B38003A	B38003B	B38009A	B38009B	B39004H
B41202A	B44001A	B44004B	B44004C	B45205A
B51003A	B55A01A	B64001A	B67001A	B67001B
B67001C	B67001D	B91003B	BC1303F	BC2001D
BC2001E	BC3005B			

CE3605A was modified to include the FORM parameter "FILE_TYPE=E" when CREATE is used, in order to create an "entry sequenced" type file rather than the default "edit" type file, so that the 360 "A"s can be written to a single line of the file.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the Tandem Ada compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Tandem Ada compiler using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a Tandem NonStop VLX operating under GUARDIAN 90, Version C10.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests was compiled and linked on the Tandem NonStop VLX, and all executable tests were run on the Tandem NonStop VLX. Results were printed from the host computer.

The compiler was tested using command scripts provided by Tandem Computers and reviewed by the validation team. The compiler was tested using all default switch settings except for the following:

Switch	Effect
SUPPRESS_LISTING	On executable tests, shortens listing to include only the error and warning messages.
EXTENDED_STACK_SIZE	Used with ADABIND (the linker) to increase the memory space available for objects at runtime.

Tests were compiled, linked, and executed (as appropriate) using a single host computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Cupertino, CA and was completed on 25 May 1988.

APPENDIX A

DECLARATION OF CONFORMANCE

Tandem Computers has submitted the following
Declaration of Conformance concerning the Tandem Ada
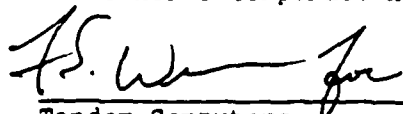compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: Tandem Computers
Ada Validation Facility: Ada Validation Facility, ASD/SCEL,
Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name: Tandem Ada   Version: Version T9270C10
Host Architecture ISA: Tandem NonStop VLX   OS&VER #: GUARDIAN 90, Version C10
Target Architecture ISA: Tandem NonStop VLX OS&VER #: GUARDIAN 90, Version C10

Implementor's Declaration

I, the undersigned, representing Tandem Computers, have implemented no
deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in
the compiler(s) listed in this declaration. I declare that Tandem
Computers is the owner of record of the Ada language compiler(s) listed
above and, as such, is responsible for maintaining said compiler(s) in
conformance to ANSI/MIL-STD-1815A. All certificates and registrations for
Ada language compiler(s) listed in this declaration shall be made only in
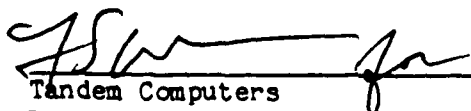the owner's corporate name.

Date: 5/25/88

Tandem Computers
Dennis McEvoy, Vice President
Software Development

Owner's Declaration

I, the undersigned, representing Tandem Computers, take full responsibility
for implementation and maintenance of the Ada compiler(s) listed above, and
agree to the public disclosure of the final Validation Summary Report. I
further agree to continue to comply with the Ada trademark policy, as
defined by the Ada Joint Program Office. I declare that all of the Ada
language compilers listed, and their host/target performance, are in
compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Date: 5/25/88

Tandem Computers
Dennis McEvoy, Vice President
Software Development

A-2

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the Tandem Ada compiler, Version T9270C10 are described in the following sections, as taken from Appendix F of the Tandem Ada documentation. All references to other sections are not references to text within this VSR, but rather, to sections of text in the Tandem Ada documentation, unless otherwise noted. Implementation-specific portions of the package STANDARD are also included in this appendix.

Contents

APPENDIX F

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

This appendix describes the implementation dependencies of Tandem Ada and is the Tandem version of the Appendix F that the Ada standard requires for each Ada reference manual. The reference manual for Tandem Ada is the *ANSI Reference Manual for the Ada Programming Language* (ANSI/MIL-STD-1815A, January 1983), plus this appendix.

This appendix discusses pragmas, attributes, packages, restrictions on representation clauses, restrictions on unchecked programming, tasking, and implementation limits, in that order.

IMPLEMENTATION-DEFINED PRAGMAS

Tandem Ada includes five implementation-defined pragmas. This subsection describes those pragmas in alphabetical order.

All five implementation-defined pragmas are for use in calling TAL subprograms. They are discussed and used in examples in Section 8, "Calling TAL Subprograms."

CONDITION_CODE Pragma

The CONDITION_CODE pragma tells the compiler to generate code
that returns a condition code to the Ada program environment when
you call the specified TAL subprogram. You can use the function
CONDITION_CODE to examine the returned condition code. You must
include this pragma for each TAL subprogram that sets a condition
code you wish to check in your Ada program.

The syntax of the CONDITION_CODE pragma is:

```
pragma CONDITION_CODE ( subprogram );


subprogram

    is the Ada subprogram name for a TAL procedure.
```

Considerations

- You must supply an INTERFACE pragma for any subprogram you
  specify in a CONDITION_CODE pragma. The INTERFACE pragma must
  precede the CONDITION_CODE pragma. If the compiler encounters
  a CONDITION_CODE pragma before it encounters a corresponding
  INTERFACE pragma, it issues a warning message and ignores the
  CONDITION_CODE pragma.

- You use the CONDITION_CODE function from the TAL_TYPES package
  to examine a condition code returned by a TAL subprogram
  to which the CONDITION_CODE pragma applies. For a full
  explanation and an example that uses the CONDITION_CODE
  function, see the discussion of "Using Condition Codes" in
  Section 8.

EXTENSIBLE_ARGUMENT_LIST Pragma

The EXTENSIBLE_ARGUMENT_LIST pragma tells the compiler that a
TAL subprogram has an extensible argument list. You must use
this pragma to declare any TAL subprogram that has an extensible
argument list.

The syntax of the EXTENSIBLE_ARGUMENT_LIST pragma is:

```
pragma EXTENSIBLE_ARGUMENT_LIST ( subprogram );


subprogram

    is the Ada subprogram name for a TAL procedure that has
    an extensible argument list.
```

Considerations

- You must supply an INTERFACE pragma for any subprogram you
  specify in an EXTENSIBLE_ARGUMENT_LIST pragma. The INTERFACE
  pragma must precede the EXTENSIBLE_ARGUMENT_LIST pragma. If
  the compiler encounters an EXTENSIBLE_ARGUMENT_LIST pragma
  before it encounters a corresponding INTERFACE pragma, it
  issues a warning message and ignores the
  EXTENSIBLE_ARGUMENT_LIST pragma.

- Ada cannot determine whether the TAL subprogram you name in
  an EXTENSIBLE_ARGUMENT_LIST pragma actually has an extensible
  argument list. If it does not, the parameter list that Ada
  generates will not correspond to the parameter list that TAL
  expects. The specific symptoms of such an error are not
  predictable.

- If you specify both an EXTENSIBLE_ARGUMENT_LIST pragma and
  a VARIABLE_ARGUMENT_LIST pragma for the same subprogram, the
  compiler ignores the VARIABLE_ARGUMENT_LIST pragma.

EXTERNAL_NAME Pragma

The EXTERNAL_NAME pragma tells the compiler that a TAL subprogram
has a TAL procedure name that can be different from the Ada
subprogram name. You must use the EXTERNAL_NAME pragma for any
TAL subprogram whose TAL procedure name is not a legal Ada name.

The syntax of the EXTERNAL_NAME pragma is:

```
pragma EXTERNAL_NAME ( Ada-name, "TAL-name" );


Ada-name

    is an Ada subprogram name for a TAL procedure.


"TAL-name"

    is the name of a TAL procedure enclosed in quotation
    marks.
```

Considerations

* You must supply an INTERFACE pragma for any subprogram you
  specify in an EXTERNAL_NAME pragma. The INTERFACE pragma must
  precede the EXTERNAL_NAME pragma. If the compiler encounters
  an EXTERNAL_NAME pragma before it encounters a corresponding
  INTERFACE pragma, it issues a warning message and ignores the
  EXTERNAL_NAME pragma.

* You should not use subprograms that have TAL names that begin
  with RSL^. The Ada run-time environment uses procedure names
  with this prefix; calling such routines might cause your Ada
  program to work incorrectly. If you specify a name that
  begins with RSL^ as the second argument to the EXTERNAL_NAME
  pragma, the compiler issues a warning message.

PRIMARY Pragma

The PRIMARY pragma tells the compiler to store the specified
objects in primary memory rather than extended memory. You must
use the PRIMARY pragma for any nonscalar object to which you
apply the attributes 'WORD_ADDR or 'STRING_ADDR. You should also
use it for any scalar object to which you apply the attributes
'WORD_ADDR or 'STRING_ADDR, though this is not required.

The syntax of the PRIMARY pragma is:

```
pragma PRIMARY ( object [ , object ] ... );


object

    is the name of a variable that is declared by an object
    declaration and that has a size known at compile time.
```

Considerations

- If you use the PRIMARY pragma for an object that is nested
  (directly or indirectly) within a procedure or task body, the
  compiler stores the object in the lower 32 KW of primary
  memory. You can always apply the attribute 'STRING_ADDR to
  such an object. If the object is word-aligned, you can also
  apply the attribute 'WORD_ADDR to the object.

- If you use the PRIMARY pragma for an object that is not
  nested within a procedure or task body (such as an object
  in a library package), the compiler stores the object in
  primary memory but not necessarily within the lower 32 KW of
  primary memory. You can apply the attribute 'WORD_ADDR to
  such an object, but applying the attribute 'STRING_ADDR may
  raise CONSTRAINT_ERROR.

- Tandem recommends that you use the PRIMARY pragma for any
  object you use with the attributes 'WORD_ADDR or 'STRING_ADDR,
  even if the compiler normally stores that object in primary
  memory. Explicitly requesting primary memory makes your
  program independent of default memory allocation, which might
  change in a later release.

VARIABLE_ARGUMENT_LIST Pragma

The VARIABLE_ARGUMENT_LIST pragma tells the compiler that a TAL
subprogram has a variable argument list. You must use this
pragma for any TAL subprogram that has a variable argument list.

The syntax of the VARIABLE_ARGUMENT_LIST pragma is:

```
pragma VARIABLE_ARGUMENT_LIST ( subprogram );


subprogram

    is the Ada subprogram name for a TAL procedure that has
    a variable argument list.
```

Considerations

- You must supply an INTERFACE pragma for any subprogram you
  specify in a VARIABLE_ARGUMENT_LIST pragma. The INTERFACE
  pragma must precede the VARIABLE_ARGUMENT_LIST pragma. If
  the compiler encounters a VARIABLE_ARGUMENT_LIST pragma before
  it encounters a corresponding INTERFACE pragma, it issues a
  warning message and ignores the VARIABLE_ARGUMENT_LIST pragma.

- Ada cannot determine whether the TAL subprogram you name in a
  VARIABLE_ARGUMENT_LIST pragma actually has a variable argument
  list. If it does not, the parameter list that Ada generates
  will not correspond to the parameter list that TAL expects.
  The specific symptoms of such an error are not predictable.

- If you specify both a VARIABLE_ARGUMENT_LIST pragma and an
  EXTENSIBLE_ARGUMENT_LIST pragma for the same subprogram, the
  compiler ignores the VARIABLE_ARGUMENT_LIST pragma.

RESTRICTIONS ON PREDEFINED PRAGMAS

Tandem Ada restricts usage of some predefined pragmas. This
subsection explains such restrictions. The restricted pragmas
are listed and discussed in alphabetical order.

CONTROLLED Pragma

The CONTROLLED pragma has no effect. Everything is controlled in
Tandem Ada.

INLINE Pragma

If you specify the INLINE pragma for a subprogram, Tandem Ada
expands calls to that subprogram inline if it can do so. If it
cannot expand a subprogram call inline, it prints a message that
explains why it cannot do so.

These are typical messages that Tandem Ada issues when it cannot
expand a subprogram call inline:

> The call to this Inline subprogram is not expanded
> because its body is not available.

> The call to this Inline subprogram is not expanded
> because its return type is unconstrained.

> The call to this Inline subprogram is not expanded
> because it is either recursive or mutually recursive.

This list does not include all possible messages of this type.

Tandem Ada can expand calls to recursive subprograms though it
does not expand a recursive subprogram's call to itself. It
never expands calls to derived subprograms.

If you use the INLINE pragma and the compiler expands a
subprogram call inline, the compilation creates a compilation
dependency on the body of the called subprogram. You must
recompile the compilation unit if you change the body of the
called subprogram.

If you use the OPTIMIZE switch for a compilation, Tandem Ada may expand some subprogram calls inline even though you did not use the INLINE pragma for the subprograms. The compiler does this only for subprograms whose bodies are included in the compilation unit, so such an expansion never creates a dependency on another compilation unit.

INTERFACE Pragma

The only language you can specify in an INTERFACE pragma is TAL.

You cannot use the INTERFACE pragma for subprograms declared within a procedure or task unit or nested within such a unit. You must declare any subprogram that you specify in an INTERFACE pragma within a library package or subpackage.

You cannot specify the INTERFACE pragma for a renamed subprogram.

See Section 8, "Calling TAL Subprograms," for detailed information about calling TAL procedures from Ada and for examples that use the INTERFACE pragma.

MEMORY_SIZE Pragma

If you use the MEMORY_SIZE pragma, the compiler issues a warning message and ignores the pragma. Tandem reserves this pragma for use in internal development.

OPTIMIZE Pragma

The OPTIMIZE pragma has no effect. If you want more than the default optimization, use the OPTIMIZE switch on the ADA compiler command, as described in Section 3.

PACK Pragma

The PACK pragma does not affect data layout. If you use it, the compiler issues a warning message and ignores the pragma.

STORAGE_UNIT Pragma

If you use the STORAGE_UNIT pragma, the compiler issues a warning
message and ignores the pragma. Tandem reserves this pragma for
use in internal development.

SUPPRESS Pragma

The SUPPRESS pragma does not affect the suppression or generation
of checking code. If you use it, the compiler issues a warning
message and ignores the pragma.

SYSTEM_NAME Pragma

If you use the SYSTEM_NAME pragma, the compiler issues a warning
message and ignores the pragma. Tandem reserves this pragma for
use in internal development.

RESTRICTIONS ON STANDARD ATTRIBUTES

Tandem Ada supports all representation attributes, though the attributes 'ADDRESS and 'STORAGE_SIZE might not have meaningful values, as explained below.

Restrictions on the 'ADDRESS Attribute

'ADDRESS returns the 32-bit extended address of an object that is not a task object. For a task object, it returns the address of the variable that contains the task identifier.

'ADDRESS returns a null address for objects that are constants whose values are known at compile time. It returns a meaningful address for other all objects.

Restrictions on the 'STORAGE_SIZE Attribute

The 'STORAGE_SIZE attribute does not return a meaningful value for access types or subtypes.

IMPLEMENTATION-DEFINED ATTRIBUTES

Tandem Ada has three implementation-defined attributes:
'EXTENDED_ADDR, 'WORD_ADDR, and 'STRING_ADDR.

- 'EXTENDED_ADDR yields the 32-bit extended address of a variable. The prefix for 'EXTENDED_ADDR must be a variable declared in an object declaration.

- 'WORD_ADDR yields the 16-bit word address of a variable. The prefix for 'WORD_ADDR must be a variable declared in an object declaration.

  'WORD_ADDR raises CONSTRAINT_ERROR if you apply it to a variable that is not stored In primary memory.

- 'STRING_ADDR yields the 16-bit string address of a variable. The prefix for 'STRING_ADDR must be a variable declared in an object declaration.

  'STRING_ADDR raises CONSTRAINT_ERROR if you apply it to a variable that is not stored in the lower 32 KW of primary memory.

All three of these attributes are for use in calling TAL subprograms from Ada. See Section 8, "Calling TAL Subprograms," for information about how to use these attributes and for examples that demonstrate their use.

STANDARD PREDEFINED PACKAGES

This section describes the implementation dependencies of the predefined packages SYSTEM, STANDARD, LOW_LEVEL_IO, TEXT_IO, DIRECT_IO, and SEQUENTIAL_IO, in that order.

SYSTEM Package

Figure F-1 lists the specifications for the predefined package SYSTEM.

```
package SYSTEM is

   type ADDRESS is private;

   type NAME is (NONSTOP);

   SYSTEM_NAME   : constant NAME := NONSTOP;

   STORAGE_UNIT : constant := 8;
   MEMORY_SIZE  : constant := 2 ** 30;

   --   System-dependent named numbers:

   MIN_INT      : constant := -9_223_372_036_854_775_808;
   MAX_INT      : constant := +9_223_372_036_854_775_807;
   MAX_DIGITS   : constant := 16;
   MAX_MANTISSA : constant := 31;
   FINE_DELTA   : constant := 2.0 ** (-31);
   TICK         : constant := 0.01;

   --   Other system-dependent declarations:

   subtype PRIORITY is INTEGER range 0 .. -1;

private
   .
   .
   .

end SYSTEM;
```

Figure F-1.   SYSTEM Package

STANDARD Package

Figure F-2 lists the specifications for the predefined package STANDARD.

```
   type SHORT_INTEGER is range -2 ** 7 .. 2 ** 7 - 1;
   for SHORT_INTEGER'SIZE use 8;

   type INTEGER is range -2 ** 15 .. 2 ** 15 - 1;
   for INTEGER'SIZE use 16;

   type LONG_INTEGER is range -2 ** 31 .. 2 ** 31 - 1;
   for LONG_INTEGER'SIZE use 32;

   type LONG_LONG_INTEGER is range -2 ** 63 .. 2 ** 63 - 1;
   for LONG_LONG_INTEGER'SIZE use 64;

   type FLOAT is digits 6 range -(2 ** 254 * (1 - 2 ** (-21)))
                              .. 2 ** 254 * (1 - 2 ** (-21));
   -- range is -FLOAT'SAFE_LARGE .. FLOAT'SAFE_LARGE
   for FLOAT'SIZE use 32;

   type LONG_FLOAT is digits 16
                 range -(2 ** 254 * (1 - 2 ** (-55))) ..
                        2 ** 254 * (1 - 2 ** (-55));
   -- range is -LONG_FLOAT'SAFE_LARGE .. LONG_FLOAT'SAFE_LARGE
   for LONG_FLOAT'SIZE use 64;

   type DURATION is delta 1 / 2 ** 14
                 range -(2 ** 31 / 2 ** 14) ..
                        (2 ** 31 - 1) / 2 ** 14;
   for DURATION'SIZE use 64;

   for BOOLEAN'SIZE use 8;

   for CHARACTER'SIZE use 8;
```

Figure F-2.   STANDARD Package

LOW_LEVEL_IO Package

Tandem Ada includes the predefined package LOW_LEVEL_IO, as
required by the Ada standard, but the subprograms in the package
do not perform input or output operations.

A call to either procedure in LOW_LEVEL_IO always returns NO_DATA
as the value of the parameter DATA. Such a call has no other
effect at run time, except for taking time and memory.

Figure F-3 lists the specifications for the LOW_LEVEL_IO package.

```
package LOW_LEVEL_IO is

    type DEVICE_TYPE is (NO_DEVICE);

    type DATA_TYPE is (NO_DATA);

    procedure SEND_CONTROL (DEVICE : DEVICE_TYPE;
                            DATA   : in out DATA_TYPE);

    procedure RECEIVE_CONTROL (DEVICE : DEVICE_TYPE;
                               DATA   : in out DATA_TYPE);
end LOW_LEVEL_IO;
```

Figure F-3. LOW_LEVEL_IO Package

TEXT_IO Package

The TEXT_IO package provides input-output operations for four
different types of disk files: EDIT files, unstructured files,
relative files, and entry-sequenced files. The default file type
for TEXT_IO is EDIT. See the *ENSCRIBE Programmer's Guide* if you
want detailed information about any of these file types.

The TEXT_IO package also provides input-output operations for
terminals and output operations for spoolers, though it does
not allow you to create either a terminal process or a spooler
process. TEXT_IO uses level 3 protocols for the first spooler
process that a program opens; it does not use level 3 protocols
for other spooler processes. See the *Spooler Programmer's Guide*
for further information about spooler processes.

Tandem Ada does not support input-output operations for processes
other than terminal and spooler processes. You can use TEXT_IO
to open another type of process for output, but TEXT_IO treats
the process as a spooler.

The maximum line length for TEXT_IO is 1320, but the actual
maximum line length for a specific file depends on the file type
and, in some cases, the contents of the line. For relative and
entry-sequenced files, the maximum line length is determined by
the record length, which can be as large as 1320. For EDIT
files, the maximum length of a line depends on the contents
of the line: any line can have up to 239 characters; lines
with appropriate contents can be longer, but 1320 characters is
the absolute maximum. (See the *EDIT User's Guide and Reference
Manual* if you need more information about the line length for
EDIT files.) Your program raises USE_ERROR if you attempt to
write a line longer than the maximum line length for the line.

The range for TEXT_IO.COUNT is 0 .. LONG_INTEGER'LAST.

The range for TEXT_IO.FIELD is 0 .. INTEGER'LAST.

There is no physical line terminator for a relative, entry-
sequenced, or EDIT file; a line is a record. The line terminator
for an odd unstructured file (an unstructured file created with
the ODDUNSTR parameter set) or an even unstructured file (an
unstructured file created without the ODDUNSTR parameter set)
that ends at an odd byte is a line feed (ASCII.LF). The line
terminator for an even unstructured file that ends at an even
byte is a double line feed (two ASCII.LF characters).

The page terminator for an unstructured file is a form feed and
a line feed (ASCII.FF followed by ASCII.LF) at the beginning of a
line. The page terminator for any other type of disk file is a
form feed (ASCII.FF) in a record by itself.

Creating Files with the TEXT_IO Package

You use the CREATE procedure to create files with TEXT_IO.
The CREATE procedure has two implementation-dependent parameters:
NAME and FORM.

For the NAME parameter, use a string that is a GUARDIAN 90 file
name. See Appendix B, "File Names," if you need information
about GUARDIAN 90 file names.

For the FORM parameter, use a string with the following syntax:

```
{ null                                                        }
{ TEXT_IO-create-option  [ , TEXT_IO-create-option ]  }
```

null

    is zero or more blanks and specifies that you want to
    use the default TEXT_IO file creation options.

TEXT_IO-create-option

    is one of the options listed below. You can specify
    creation options in any order, but you can only specify
    each option once.

DATA_BLOCKLEN = *block-length*

    specifies the block length for the new file. *block-length* must be an integer in the range 1 to 4096. The
    actual block length for a structured file must be a
    multiple of 512, but Ada automatically rounds up to an
    appropriate value. The default block length is 1024.

FILE_CODE = *code-number*

    specifies the operating system file code for the new
    file. *code-number* must be an integer in the range 0 to
    65,535. You cannot use this option for an EDIT file;
    Ada always uses file code 101. For other file types,
    the default is 0.

$\longrightarrow$

PRIMARY_EXTENT_SIZE = *primary-extent-size*

   specifies the size of the primary extent for the new
   file. *primary-extent-size* must be an integer in the
   range 0 to 65,535. The default is 4.

SECONDARY_EXTENT_SIZE = *secondary-extent-size*

   specifies the size of the secondary extents for the new
   file. *secondary-extent-size* must be an integer in the
   range 0 to 65,535. The default is 16.

FILE_TYPE = *type-code*

   specifies the file type for the new file, as follows:

type-code	File Type	
D	EDIT	EDIT is the default.
U	Unstructured	
R	Relative	
E	Entry-sequenced	

RECORDLEN = *record-length*

   specifies the maximum record length for a new relative
   or entry-sequenced file. *record-length* must be an inte-
   ger in the range 1 to 1320. You cannot use this option
   for an EDIT or unstructured file. The default is 132
   for a relative file and 1320 for an entry-sequenced
   file.

ODDUNSTR

   specifies that the new file allows reading and writing
   of odd-numbered byte counts and positioning to odd-
   numbered byte addresses. You can use this option only
   for an unstructured file. The default is to create a
   file that works only on even-numbered byte counts.

Considerations for Creating Files With TEXT_IO

- Your program raises USE_ERROR if you attempt to create a file with the same file name as an existing file, if you attempt to create a file for input, if you attempt to create a terminal or a spooler, or if you specify an incorrect FORM string.

- The CREATE procedure always opens a new file in EXCLUSIVE mode. No other process can read or write the file until your program closes it.

Examples of Creating Files With TEXT_IO

- This example creates an EDIT file named MYFILE on the same system, volume, and subvolume as the executing program.

  ```
  TEXT_IO.CREATE ( FILE => MYFILEVAR, NAME => "MYFILE");
  ```

- This example creates a relative file named MARCH on the subvolume named SALES on the same system and volume as the executing program. The new file has a record length of 200 and also has larger primary and secondary extents than normal.

  ```
  TEXT_IO.CREATE ( FILE => SALESFILE,
                   MODE => OUT_FILE,
                   NAME => "SALES.MARCH",
                   FORM => "PRIMARY_EXTENT_SIZE = 10,
                            SECONDARY_EXTENT_SIZE = 40,
                            RECORDLEN = 200,
                            FILE_TYPE = R");
  ```

- This example creates an unstructured file named EMPFILE that can be accessed from odd-numbered byte positions. The file is located on subvolume PAYROLL, volume $HR of the \HDQ system. The block length for the new file is 4096, since Ada rounds up the specified block length to the nearest multiple of 512.

  ```
  TEXT_IO.CREATE ( FILE => EMP,
                   MODE => OUT_FILE,
                   NAME => "\HDQ.$HR.PAYROLL.EMPFILE",
                   FORM => "FILE_TYPE = U, ODDUNSTR,
                   DATA_BLOCKLEN = 4000");
  ```

DIRECT_IO Package

The DIRECT_IO package provides input-output operations for relative disk files. See the *ENSCRIBE Programmer's Guide* if you want detailed information about relative disk files.

The DIRECT_IO package determines the record length for a file based on the size of the objects for the file. As a result, you cannot instantiate DIRECT_IO for an unconstrained type, except for a record that has discriminants with default expressions. In that case, DIRECT_IO uses the record length needed for the largest possible object of the type.

The range for DIRECT_IO.COUNT is 0 .. LONG_INTEGER'LAST.

Your program raises DATA_ERROR if you attempt to read a nonexistent record from a DIRECT_IO file.

Creating Files With the DIRECT_IO Package

Use the CREATE procedure to create files with DIRECT_IO. The CREATE procedure has two implementation-dependent parameters: NAME and FORM.

For the NAME parameter, use a string that is a GUARDIAN 90 file name. See Appendix B, "File Names," if you need information about GUARDIAN 90 file names.

For the FORM parameter, use a string with the following syntax:

```
{ null                                                      }
{ DIRECT_IO-create-option  [ , DIRECT_IO-create-option ]  }
```

null

   is zero or more blanks. A null form string specifies
   that you want to use the default creation options.

DIRECT_IO-create-option

   is one of the creation options described below. You can
   specify creation options in any order, but you can only
   specify each option once.

DATA_BLOCKLEN = *block-length*

   specifies the block length for the new file. *block-
   length* must be an integer in the range 1 to 4096. The
   actual block length is always a multiple of 512, but Ada
   rounds up to an appropriate value. The default block
   length is 1024.

FILE_CODE = *code-number*

   specifies the operating system file code for the new
   file. *code-number* must be an integer in the range 0 to
   65,535. The default is 0.

PRIMARY_EXTENT_SIZE = *primary-extent-size*

   specifies the size of the primary extent for the new
   file. *primary-extent-size* must be an integer in the
   range 0 to 65,535. The default is 4.

SECONDARY_EXTENT_SIZE = *secondary-extent-size*

   specifies the size of the secondary extents for the new
   file. *secondary-extent-size* must be an integer in the
   range 0 to 65,535. The default is 16.

Considerations for Creating Files With DIRECT_IO

* Your program raises USE_ERROR if you attempt to create a file with the same file name as an existing file, if you attempt to create a file with a record length of zero, if you attempt to create a file for input, or if you specify an incorrect FORM string.

* The CREATE procedure always opens a new file in EXCLUSIVE mode. No other process can read or write the file until your program closes it.

Examples of Creating Files With DIRECT_IO

* This example creates a relative file named RELFILE on the same system, volume, and subvolume as the executing program.

  ```
  DIRECT_IO.CREATE ( FILE => RELFILEVAR, NAME => "RELFILE");
  ```

* This example creates a relative file named TAXFILE that has a block length of 2048 and a file code of 25. The file is located on subvolume PAYROLL, volume $HR of the \HDQ system.

  ```
  DIRECT_IO.CREATE ( FILE => TF,
                     MODE => OUT_FILE,
                     NAME => "\HDQ.$HR.PAYROLL.TAXFILE",
                     FORM => "DATA_BLOCKLEN = 2048,
                             FILE_CODE=25");
  ```

SEQUENTIAL_IO Package

The SEQUENTIAL_IO package provides input-output operations for entry-sequenced disk files. See the *ENSCRIBE Programmer's Guide* if you want detailed information about entry-sequenced disk files.

The SEQUENTIAL_IO package determines the record length for a file based on the size of the objects for the file. As a result, you cannot instantiate SEQUENTIAL_IO for an unconstrained type, except for a record that has discriminants with default expressions. In that case, SEQUENTIAL_IO uses the record length needed for the largest possible object of the type.

Creating Files With the SEQUENTIAL_IO Package

You use the CREATE procedure to create files with SEQUENTIAL_IO. The CREATE procedure has two implementation-dependent parameters: NAME and FORM.

For the NAME parameter, use a string that is a GUARDIAN 90 file name. See Appendix B, "File Names," if you need information about GUARDIAN 90 file names.

For the FORM parameter, use a string with the following syntax:

```
{ null                                                    }
{ SEQ_IO-create-option  [ , SEQ_IO-create-option ]  }
```

null

    is zero or more blanks. A null form string specifies
that you want to use the default creation options.

SEQ_IO-create-option

    is one of the creation options described below. You can
specify creation options in any order, but you can only
specify each option once.

DATA_BLOCKLEN = *block-length*

    specifies the block length for the new file. *block-length* must be an integer in the range 1 to 4096. The
actual block length is always a multiple of 512, but Ada
rounds up to an appropriate value. The default is 1024.

FILE_CODE = *code-number*

    specifies the operating system file code for the new
file. *code-number* must be an integer in the range 0 to
65,535. The default is 0.

PRIMARY_EXTENT_SIZE = *primary-extent-size*

    specifies the size of the primary extent for the new
file. *primary-extent-size* must be an integer in the
range 0 to 65,535. The default is 4.

SECONDARY_EXTENT_SIZE = *secondary-extent-size*

    specifies the size of the secondary extents for the new
file. *secondary-extent-size* must be an integer in the
range 0 to 65,535. The default is 16.

Considerations for Creating Files With SEQUENTIAL_IO

- Your program raises USE_ERROR if you attempt to create a file with the same file name as an existing file, if you attempt to create a file with a record length of zero, if you attempt to create a file for input, or if you specify an incorrect FORM string.

- The CREATE procedure always opens a new file in EXCLUSIVE mode. No other process can read or write the file until your program closes it.

Examples of Creating Files With SEQUENTIAL_IO

- This example creates an entry-sequenced file named LOGFILE on the same system, volume, and subvolume as the executing program.

  ```
  CREATE ( FILE => LOGFILE, NAME => "LOGFILE");
  ```

- This example creates an entry-sequenced file named DATA that has a block length of 2048 and a primary extent size of 32. The file is located on subvolume DALLAS, volume $TEXAS of the \US system.

  ```
  SEQUENTIAL_IO.CREATE ( FILE => FILEVAR,
                         MODE => OUT_FILE,
                         NAME => "\US.$TEXAS.DALLAS.DATA",
                         FORM => "DATA_BLOCKLEN = 2048,
                                  PRIMARY_EXTENT_SIZE = 32");
  ```

Opening Files With TEXT_IO, DIRECT_IO, or SEQUENTIAL_IO

You use the OPEN procedure to open files with any I/O package.
The OPEN procedure has two implementation-dependent parameters:
NAME and FORM.

For the NAME parameter, use a string that is a GUARDIAN 90 file
name. See Appendix B, "File Names," if you need information
about GUARDIAN 90 file names.

For the FORM parameter, use a string with the following syntax:

```
{   SHARED      }
{   EXCLUSIVE   }
{   PROTECTED   }
{   null        }
```

SHARED

   specifies that other processes can read or write the
   file while your process has it open. You cannot use
   SHARED for a DIRECT_IO file with mode INOUT_FILE or
   OUT_FILE.

EXCLUSIVE

   specifies that other processes cannot read or write the
   file while your process has it open. You cannot use
   EXCLUSIVE for a terminal or spooler.

PROTECTED

   specifies that other processes can read the file while
   your process has it open, but cannot write to it. You
   cannot use PROTECTED for a terminal, a spooler, or a
   DIRECT_IO file with mode INOUT_FILE or OUT_FILE.

null

   is zero or more blanks and specifies that you want to
   use the default for the type of file you are opening.

Considerations for Calls to OPEN

- The default for a disk file with mode IN_FILE is SHARED. The default for a disk file with mode OUT_FILE or mode INOUT_FILE is EXCLUSIVE. The default for a terminal or spooler is SHARED.

- Your program raises USE_ERROR if you make any error in the FORM string parameter.

- You cannot open a DIRECT_IO file or a SEQUENTIAL_IO file with a record length of zero or with a record length that is different from the record length you used to instantiate the package. Your program raises USE_ERROR if you attempt to do so.

- You cannot use TEXT_IO to open an entry-sequenced or relative file with a maximum record length greater than 1320. Your program raises USE_ERROR if you attempt to do so.

- When you open an existing TEXT_IO or SEQUENTIAL_IO file with mode OUT_FILE, Ada deletes the file and re-creates it. The new file has the same characteristics (block size, record type, and so forth) as the original file but contains no data.

Examples of Calls to OPEN

- This example opens a file named DATA on the same system, volume, and subvolume as the executing program. Other programs can read or write to the file while this program has it open.

```
OPEN ( FILE => FILEVAR,
       MODE => IN_FILE,
       NAME => "DATA",
       FORM => "SHARED");
```

- This example opens a file named DATA located in subvolume PAYROLL, volume $PERS, on system \HDQ. Other programs can read the file while this program has it open, but they cannot write to it.

```
OPEN ( FILE => TAXFILE,
       MODE => IN_OUT_FILE,
       NAME => "\HDQ.$PERS.PAYROLL.TAXES",
       FORM => "PROTECTED");
```

Resetting Files

You can reset any type of file to mode IN_FILE, but the
reset does not change the exclusion mode (SHARED, EXCLUSIVE, or
PROTECTED) in effect for the file.

If you reset a DIRECT_IO file to mode INOUT_FILE or mode
OUT_FILE, Ada closes the file and reopens it with an EXCLUSIVE
exclusion mode.

You cannot reset a TEXT_IO or SEQUENTIAL_IO file to mode
OUT_FILE. If you attempt to do so, your program raises
USE_ERROR.

Closing Files

Your program should close every file that it explicitly opens.
If you fail to close a file, you can leave it in an inconsistent
state, especially if it is an EDIT file.

Standard Input and Output Files

Ada automatically opens and closes the standard input and output
files using the TEXT_IO package. By default, both files are
the home terminal for your program. You can change this by
specifying other file names with the IN and OUT parameters of the
RUN command that starts your program, as described in Section 5.

If you specify a standard input file that does not exist or
cannot be opened, Ada sends this message to the home terminal:

    Cannot open Standard Input File.

If you specify a standard output file that does not exist,
TEXT_IO creates a new file of that name, using the default values
of TEXT_IO CREATE. If you specify a file that does exist,
TEXT_IO deletes the file and re-creates it with the original
characteristics. If the file cannot be created (or deleted and
re-created) for some reason, Ada sends this message to the home
terminal:

    Cannot create Standard Output File.

You program continues to execute even if Ada cannot open
the standard input and output files, but the program raises
STATUS_ERROR if you attempt to read or write to an unopened file.

ADDITIONAL, TANDEM-DEFINED PACKAGES

Tandem Ada includes four predefined packages in addition to
the standard packages with implementation dependencies. The
additional packages are:

* BIT_OPERATIONS

* COMMAND_INTERPRETER_INTERFACE

* TAL_TYPES

* SYSTEM_CALLS

This subsection describes these packages in alphabetical order.

BIT_OPERATIONS Package

The BIT_OPERATIONS package provides Tandem Ada programs with bit-
manipulation capabilities similar to those of TAL. You can use
BIT_OPERATIONS in any Tandem Ada program but it is primarily
intended for use in programs that call TAL procedures that use
unsigned quantities.

The BIT_OPERATIONS package is described in detail in the subsec-
tion "Bit Operations" in Section 8, "Calling TAL Subprograms."

COMMAND_INTERPRETER_INTERFACE Package

The COMMAND_INTERPRETER_INTERFACE package provides Tandem Ada
programs with the ability to read information from STARTUP,
ASSIGN, and PARAM messages sent to the executing program by the
operating system command interpreter. Section 5, "Running Ada
Programs," discusses these messages. For additional information
about them, see the *GUARDIAN 90 Operating System Programmer's
Guide*.

The TEXT_IO, DIRECT_IO, and SEQUENTIAL_IO packages name the
COMMAND_INTERPRETER_INTERFACE package in a with clause and au-
tomatically read the STARTUP, ASSIGN, and PARAM messages. Your
program can also name the package in a with clause and use
the subprograms in the package to read these messages. The
elaboration code in the package reads the messages, so if you use
the package in the elaboration of another compilation unit, the
dependent unit must specify the ELABORATE pragma for the package.

The COMMAND_INTERPRETER_INTERFACE package includes types, exceptions, and subprograms. Figure F-4 lists the specifications for the types and exceptions in the package. Figure F-5 lists the specifications for subprograms that read the STARTUP message. Figure F-6 lists the specifications for subprograms that read the ASSIGN message. Figure F-7 lists the specifications for subprograms that read the PARAM message.

```ada
package COMMAND_INTERPRETER_INTERFACE is

   CANT_READ_MESSAGES : exception;
   -- Raised by all routines if the Ada process could not
   -- read the command interpreter messages.

   FIELD_NOT_PRESENT  : exception;
   -- Raised when a field selection of an assign message is
   -- absent.

   type ASSIGN_MESSAGE_T is private;

   NO_ASSIGN : constant ASSIGN_MESSAGE_T;


   type FILE_EXCLUSION_T is (SHARED, EXCLUSIVE, PROTECTED);

   type FILE_ACCESS_T is (IN_OUT, INPUT, OUTPUT);

   subtype LOGICAL_FILENAME_T is STRING (1 .. 31);

   type PARAM_MESSAGE_T is private;

   NO_PARAM : constant PARAM_MESSAGE_T;
```

Figure F-4.   Exceptions and Types From the
               COMMAND_INTERPRETER_INTERFACE
               Package

```
function GET_DEFAULT return STRING;
-- Returns the default volume and subvolume specified by
-- the startup message in the form $VOL.SUBVOL.

function GET_INFILE return STRING;
-- Returns the IN file specified by the startup message
-- in the form $VOL.SUBVOL.DNAME.

function GET_OUTFILE return STRING;
-- Returns the OUT file specified by the startup
-- message in the form $VOL.SUBVOL.DNAME.

function GET_STARTUP_MESSAGE_PARAM return STRING;
-- Returns the parameter string specified in the RUN
-- command line from the startup message.  The returned
-- string does not include any trailing null characters
-- with which the command interpreter padded the string.
```

Figure F-5. Subprograms to Read the STARTUP Message

```
procedure ASSIGN_LIST_RESET;
-- Resets the pointer to the first assign message.

function GET_NEXT_ASSIGN return ASSIGN_MESSAGE_T;
-- Returns the next message from the assign message list
-- or, if no message is left, returns NO_ASSIGN.

function SEARCH_ASSIGN (PROG_NAME : in STRING;
                        FILE_NAME : in STRING)
                        return ASSIGN_MESSAGE_T;
-- Searches the list of assign messages for the logical
-- unit specified.  A match occurs when both the input
-- program name and file name are identical to those of
-- an assign message.  Otherwise, the function returns
-- NO_ASSIGN.
```

Figure F-6. Subprograms to Read ASSIGN Messages (Page 1 of 4)

```
procedure GET_LOGICAL_UNIT_NAMES
                    (ASSIGN : in ASSIGN_MESSAGE_T;
                     PROG_NAME : out LOGICAL_FILENAME_T;
                     PROG_NAME_LEN : out INTEGER;
                     FILE_NAME : out LOGICAL_FILENAME_T;
                     FILE_NAME_LEN : out INTEGER);
-- Returns the program name and file name of the logical
-- unit for the specified assign message.

function IS_TANDEM_FILENAME_PRESENT
            (ASSIGN : ASSIGN_MESSAGE_T) return BOOLEAN;
-- Returns TRUE if the file name is present;
-- returns FALSE otherwise.

function IS_PRI_EXTENT_PRESENT (ASSIGN : ASSIGN_MESSAGE_T)
                                    return BOOLEAN;
-- Returns TRUE if the primary extent is present;
-- returns FALSE otherwise.

function IS_SEC_EXTENT_PRESENT (ASSIGN : ASSIGN_MESSAGE_T)
                                    return BOOLEAN;
-- Returns TRUE if the secondary extent is present;
-- returns FALSE otherwise.

function IS_FILECODE_PRESENT (ASSIGN : ASSIGN_MESSAGE_T)
                                    return BOOLEAN;
-- Returns TRUE if the file code is present;
-- returns FALSE otherwise.

function IS_EXCLUSION_PRESENT (ASSIGN : ASSIGN_MESSAGE_T)
                                    return BOOLEAN;
-- Returns TRUE if the exclusion spec is present;
-- returns FALSE otherwise.

function IS_ACCESS_SPEC_PRESENT
                        (ASSIGN : ASSIGN_MESSAGE_T)
                                return BOOLEAN;
-- Returns TRUE if the access spec is present;
-- returns FALSE otherwise.
```

Figure F-6.   Subprograms to Read ASSIGN Messages (Page 2 of 4)

```
function IS_RECORD_SIZE_PRESENT
                            (ASSIGN : ASSIGN_MESSAGE_T)
                                    return BOOLEAN;
-- Returns TRUE if the record size is present;
-- returns FALSE otherwise.

function IS_BLOCK_SIZE_PRESENT
                            (ASSIGN : ASSIGN_MESSAGE_T)
                                    return BOOLEAN;
-- Returns TRUE if the block size is present;
-- returns FALSE otherwise.

function GET_TANDEM_FILENAME (ASSIGN : ASSIGN_MESSAGE_T)
                                    return STRING;
-- Returns the operating system file name for the
-- specified assign message; raises FIELD_NOT_PRESENT
-- if the field is absent.

function GET_PRI_EXTENT (ASSIGN : ASSIGN_MESSAGE_T)
                            return INTEGER;
-- Returns the primary extent for the specified
-- assign message; raises FIELD_NOT_PRESENT if the
-- field is absent.

function GET_SEC_EXTENT (ASSIGN : ASSIGN_MESSAGE_T)
                            return INTEGER;
-- Returns the secondary extent for the specified
-- assign message; raises FIELD_NOT_PRESENT if the
-- field is absent.

function GET_FILECODE (ASSIGN : ASSIGN_MESSAGE_T)
                        return INTEGER;
-- Returns the file code for the specified
-- assign message; raises FIELD_NOT_PRESENT if the
-- field is absent.

function GET_EXCLUSION (ASSIGN : ASSIGN_MESSAGE_T)
                        return FILE_EXCLUSION_T;
-- Returns the exclusion specification for the
-- specified assign message; raises FIELD_NOT_PRESENT
-- if the field is absent.
```

Figure F-6.   Subprograms to Read ASSIGN Messages (Page 3 of 4)

```
function GET_ACCESS_SPEC (ASSIGN : ASSIGN_MESSAGE_T)
                          return FILE_ACCESS_T;
-- Returns the access specification for the
-- specified assign message; raises FIELD_NOT_PRESENT
-- if the field is absent.

function GET_RECORD_SIZE (ASSIGN : ASSIGN_MESSAGE_T)
                          return INTEGER;
-- Returns the record size for the specified
-- assign message; raises FIELD_NOT_PRESENT if the
-- field is absent.

function GET_BLOCK_SIZE (ASSIGN : ASSIGN_MESSAGE_T)
                          return INTEGER;
-- Returns the block size for the specified
-- assign message; raises FIELD_NOT_PRESENT if the
-- field is absent.
```

Figure F-6.   Subprograms to Read ASSIGN Messages (Page 4 of 4)

```
procedure PARAM_LIST_RESET;
-- Resets the pointer to the beginning of the param
-- message list.

function GET_NEXT_PARAM return PARAM_MESSAGE_T;
-- Returns the next message from the param message
-- list; returns NO_PARAM if no message is left.

function SEARCH_PARAM_LIST (NAME : STRING)
                          return PARAM_MESSAGE_T;
-- Searches the param message list for a param with the
-- specified name and returns the message for that param;
-- returns NO_PARAM if it can't find a match.

function GET_PARAM_NAME (PARAM : PARAM_MESSAGE_T)
                          return STRING;
-- Returns the param name of the specified param message.

function GET_PARAM_VALUE (PARAM : PARAM_MESSAGE_T)
                          return STRING;
-- Returns the value of the specified param message.
```

Figure F-7.   Subprograms to Read the PARAM Messages

SYSTEM_CALLS Package

The SYSTEM_CALLS package contains Ada subprogram specifications for many GUARDIAN 90 operating system procedures. You can use SYSTEM_CALLS to save the trouble of writing these declarations yourself if you plan to call these procedures from Ada.

See "Using GUARDIAN 90 Procedures in the SYSTEM_CALLS Package" in Section 8 for a more detailed explanation of the contents of this package.

TAL_TYPES Package

The TAL_TYPES package defines types, subtypes, and functions for use in Ada programs that call TAL subprograms.

See Section 8, "Calling TAL Subprograms," for an explanation of the contents of TAL_TYPES and for examples that demonstrate how to use TAL_TYPES to call TAL subprograms from Ada.

Figure F-8 lists the specifications for the TAL_TYPES package.

```
package TAL_TYPES is

     type CONDITION_CODE_T is (CCG, CCE, CCL);

     subtype STRING is SHORT_INTEGER;
     subtype INT is INTEGER;
     subtype INT_32 is LONG_INTEGER;
     subtype FIXED is LONG_LONG_INTEGER;

     type STRING_ADDR is limited private;
     type WORD_ADDR is limited private;
     type EXTENDED_ADDR is limited private;

     generic
          type RESULT_TYPE is limited private;
     function NOT_SPECIFIED return RESULT_TYPE;

     function CONDITION_CODE return CONDITION_CODE_T;

private

   type STRING_ADDR is new INTEGER;
   type WORD_ADDR is new INTEGER;
   type EXTENDED_ADDR is new LONG_INTEGER;

  end TAL_TYPES;
```

Figure F-8. The TAL_TYPES Package

RESTRICTIONS ON REPRESENTATION CLAUSES

In addition to the rules for using representation clauses
and representation pragmas described in the *ANSI Reference
Manual for the Ada Programming Language*, Tandem Ada restricts
size specifications in length clauses, record representation
clauses, address clauses, enumeration representation clauses,
the specification of 'SMALL for fixed-point types, and the
specification of 'STORAGE_SIZE. This subsection describes these
restrictions.

Restrictions on Size Specifications in Length Clauses

For records and arrays, the value of the static expression in
a length clause for T'SIZE must be a multiple of 8 and of the
alignment. Also, the value must be at least as large as the
default size the compiler calculates for T.

Table F-1 shows the possible values of N for different data types
in the "for T'SIZE use N" clause. For scalar types, just a few
values are valid. You cannot use the length clause for access
types.

Table F-1. Size Specifications for Different Types

Type	Possible Values of T'SIZE
Integer	8, 16, 32, and 64
Enumeration	8 and 16
Fixed point	64
Floating point	32 and 64
Task	32
Composite	Multiples of 8 and of the alignment
Access	Not supported

If a representation clause increases the size of a type, then
the compiler creates some filler space. For scalar types, which
the compiler always stores right-justified within a container,
the filler space is sign-extended. For composite types, which
the compiler always stores left-justified within a container, the
filler space is zero-filled.

A subtype of a type typically has the same size as the type,
however, a constrained record subtype can be smaller than the
record type. If you specify a length clause for the record type,
the compiler uses the length in the clause to allocate space
for the type. But when allocating space for an object of a
constrained subtype, the compiler ignores the length clause for
the type and chooses a size for the subtype.

Restrictions on Record Representation Clauses

The restrictions on component clauses ("at N range L .. R") are:

- The compiler must be able to determine the size of the
  component subtype at compile time, as explained in "Sizes the
  Compiler Knows at Compile Time," in Appendix C.

- The size of the range you specify (R - L + 1) must equal the
  size of the component type.

- The value for L must be a multiple of 8 and the component must
  begin on a byte boundary.

- All values supplied for a record component offset must be
  nonnegative (N * 8 + L >= 0).

- Components from a variant part must follow components from the
  fixed part in the record layout.

The compiler's layout algorithm implies some additional restric-
tions. For a description of the algorithm, see "Complex Records"
under "Record Types," in Appendix C.

The restrictions on alignment clauses ("at mod N") are:

- The value of N must be at least as large as the default
  alignment the compiler chose for the record.

- The value of N must be either 1 or 2 bytes.

Tandem Ada does not support record representation clauses for
records that contain generic formals.

Restrictions on Address Clauses

The Tandem Ada compiler does not support address clauses.

Restrictions on Enumeration Representation Clauses

The Tandem Ada compiler does not support enumeration representation clauses.

Restrictions on Specification of 'SMALL for Fixed-Point Types

The value of the static expression in a length clause for 'SMALL must be a power of 2 in the following range:

    2 ** (-255) to 2 ** 255

The value specified for 'SMALL must be in the range precisely represented by the positive range of the predefined types FLOAT and LONG_FLOAT. As implied in the *ANSI Reference Manual for the Ada Programming Language*, the value must also satisfy the relation:

    max (ceiling (log2 (abs (LB) / small)),
         ceiling (log2 (abs (UB) / small))) <= SYSTEM.MAX_MANTISSA

In other words, the number of binary digits in the mantissa of the model numbers for fixed-point types must be less than or equal to the maximum number of binary digits, SYSTEM.MAX_MANTISSA, which is 31.

Restrictions on Specification of 'STORAGE_SIZE

For tasks, the value you specify for 'STORAGE_SIZE must be greater than 0 but less than 2 ** 27 bytes. The default is 2 ** 18 bytes. For a description of how tasks use memory, see Appendix E, "Memory Usage on NonStop Systems."

Tandem Ada does not support 'STORAGE_SIZE for access types.

RESTRICTIONS ON UNCHECKED PROGRAMMING

The generic function UNCHECKED_CONVERSION has these restrictions:

- The sizes of the source and target types must be the same.

- The sizes of the source and target types must be known at compile time. (For information about this, see "Sizes the Compiler Knows at Compile Time," in Appendix C.)

- The source and target types must not be unconstrained records or arrays.

Tandem Ada supports the generic procedure UNCHECKED_DEALLOCATION, but it does not actually reclaim memory space, even though it resets an access value to null. Tandem Ada reclaims space that it allocates for temporary variables that it creates for subprogram calls; it does not reclaim space for data that a program creates directly.

TASKING

For the most part, Tandem Ada executes parallel tasks sequentially, interleaving the execution of various tasks as appropriate for the program. Parallel tasks execute in parallel, however, when a program performs certain input-output operations.

Tasks that perform input-output operations using the TEXT_IO package, the SEQUENTIAL_IO package, the DIRECT_IO package, or the SYSTEM_CALLS package can execute in parallel with other tasks during the input-output operations. Except for operations on EDIT files, which always execute sequentially, input-output operations from these packages execute in parallel with another task whenever possible. For example, while one task is waiting for input from a terminal, which can take a long time, a parallel task can execute.

An input-output operation appears indivisible to the task that executes it, even if the operation executes in parallel with another task, and the task that executes the input-output operation does not continue until the operation completes. As a result, programmers generally do not need to consider the parallelism when they code individual tasks. The only special consideration imposed by the implementation of parallel input-output involves calls to the operating system procedures AWAITIO and AWAITIOX.

Tandem Ada implements parallel processing for input-output opera-
tions by using nowait input-output operations. Calls to the
operating system procedure AWAITIO or AWAITIOX with a file
parameter of -1 can interfere with outstanding nowait input-
output operations. The specific symptoms of such an interference
are impossible to predict.

Consequently, you should not call AWAITIO or AWAITIOX with a
file parameter of -1 in tasks that can execute in parallel with
tasks that use the TEXT_IO package, the SEQUENTIAL_IO package,
the DIRECT_IO package, or the SYSTEM_CALLS package to perform
input-output operations on files other than EDIT files.

If you want more information about nowait input-output
operations, see the *GUARDIAN 90 Operating System Programmer's
Guide*.

IMPLEMENTATION LIMITS

Table F-2 lists some Tandem Ada limits on the use of language
features.

Table F-2.   Implementation Limits

Language Feature	Maximum Number
Characters in an identifier	200
Characters in a line	200
Discriminants in a constraint	256
Associations in a record aggregate	256
Fields in a record aggregate	256
Formal parameters in a generic unit	256
Nested contexts	250
Bytes for an object	~2 ** 27
Words of object code for a subprogram	32767
Library units in a program	500
Compilation units and subprograms in a program (The compiler reserves approximately 1000 entries for run-time routines.)	~15000
Units named in a compilation unit's with clauses	255
Dynamic components in a record	256
Array dimensions	7
Control statement nesting level	256
Literals for an enumeration type	32767
Tasks for a program	32767
Entries for a task	32767
Subprogram nesting level in a calling sequence (For example, f(f(f(x))) has three nesting levels.)	256
Unique strings and identifiers for a compilation unit	4096

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such
as the maximum length of an input line and invalid file names. A test that
makes use of such values is identified by the extension .TST in its file
name. Actual values to be substituted are represented by names that begin
with a dollar sign. A value must be substituted for each of these names
before the test is run. The values used for this validation are given
below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..199 => 'A', 200 => '1')
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..199 => 'A', 200 => '2')
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..99 | 101..200 => 'A', 100 => '3')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..99 | 101..200 => 'A', 100 => '4')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..197 => '0', 198..200 => "298")

TEST PARAMETERS

Name and Meaning	Value
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..194 => '0', 195..200 => "69.0E1")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1..100 => 'A')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1..99 => 'A', 100 => '1')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..180 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
.$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	32767
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	X}]!@
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	XYZ*
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0

Name and Meaning	Value
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	100_000_000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	bad-character#^
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	muchtoolongname
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32768
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32767
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32768
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-100_000_000.0
$MAX_DIGITS Maximum digits supported for floating-point types.	16
$MAX_IN_LEN Maximum input line length permitted by the implementation.	200
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	9223372036854775807
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	9223372036854775808

TEST PARAMETERS

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 =>"2:", 3..197 =>'0', 198..200 =>"11:")
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 =>"16:", 4..196 =>'0', 197..200 =>"F.E:")
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 => '"', 2..199 => 'A', 200 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-9223372036854775808
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	LONG_LONG_INTEGER
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFF_FFFF_FFFE#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- B28003A: A basic declaration (line 36) incorrectly follows a later declaration.

- E28005C: This test requires that "PRAGMA LIST (ON);" not appear in a listing that has been suspended by a previous "PRAGMA LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the AJPO.

- C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT_ERROR.

- C35502P: The equality operators in lines 62 and 69 should be inequality operators.

- A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

- C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

- C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may, in fact, raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

- C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.

- C37213H: The subtype declaration of SCONS in line 100 is incorrectly expected to raise an exception when elaborated.

- C37213J: The aggregate in line 451 incorrectly raises CONSTRAINT_ERROR.

- C37215C, C37215E, C37215G, and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type CONS.

- C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

- C41402A: The attribute 'STORAGE_SIZE is incorrectly applied to an object of an access type.

- C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

- C45614C: The function call of IDENT_INT in line 15 uses an argument of the wrong type.

- A74106C, C85018B, C87B04B, and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.

- BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.

- AD1A01A: The declaration of subtype SINT3 raises CONSTRAINT_ERROR for implementations which select INT'SIZE to be 16 or greater.

- CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.

- CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.